An Integrated Test Generation Tool for Enhanced Coverage of Simulink/Stateflow Models

P. Peranandam*, S. Raviram†, M. Satpathy*, A. Yeolekar*, A. Gadkari*,S. Ramesh*

* India Science Lab, General Motors Global R&D, GM Tech Center (India), Bangalore 560066
† General Motors Powertrain – India, GM Tech Center (India), Bangalore 560066
Email: {prakash.peranandam, sachin.raviram, manoranjan.satpathy, ramesh.s}@gm.com
{ambar.gadkari, avyeolekar}@gmail.com

***Abstract*—Simulink/Stateflow (SL/SF) is the primary modeling notation for the development of control systems in automotive and aerospace industries. In model based testing, test cases derived from a design model are used to show model-code conformance. Safety standards such as ISO 26262 recommend model based testing to show the conformance of a software with the corresponding model. From our experiments with various test generation techniques, we have observed that their coverage capabilities are complementary in nature. With this observation in mind, we have developed a new tool called `SmartTestGen` which integrates different test generation techniques. In this paper, we discuss `SmartTestGen` and the different test generation techniques utilized – random testing, constraint solving, model checking and heuristics. We experimented with 20 production-quality SL/SF models and compared the performance of our tool with that of two prominent commercial tools.**

I. Introduction

In model based testing, models form the basis for generating test cases which can be used to show model-code conformance. We have performed experiments with many test case generation techniques such as, model checking, random testing, local constraint solving (combination of random testing and constraint solving) – on a number of industrial-strength SL/SF models. We have also experimented with the heuristics based guided coverage technique as in [1] which in particular is effective in covering deep targets and targets involving non-linear constraints. We observed that the coverages achieved by the individual techniques in a broader sense complement each other. Thus there is a case for integrating the different techniques. With this aim in mind, we have developed `SmartTestGen (STGen)`, an integrated test generation tool which uses various test generation engines, each engine implementing a different technique. The test engines are invoked in a particular order – a different ordering is possible though – so that cheap targets are covered early by a cheaper engine. We have considered 20 production-quality SL/SF models from industry and observed that `STGen` outperforms two prominent commercial tools.

The main contributions of our work are:

- Design and implementation of an integrated test case generation tool which integrates multiple test case generation engines, all developed by us.
- Evaluation of our prototype tool over 20 engineering models from the automotive domain and comparison of our results with those of two commercial tools.

II. State Of The Art Tools

Some of the prominent commercially available tools for generating test cases from SL/SF models are: *Reactis* from Reactive Systems Inc. [2], *Embedded Tester* from BTC [3] and *Simulink Design Verifier (SDV)* from Mathworks [4]. The *Reactis tester* uses a combination of random testing and guided simulation. In *Embedded tester*, SL/SF models are first fed to the `TargetLink` code generator which produces C-code. Analysis on the generated C-code is used to produce test cases. The main functions of *SDV* [4] are test case generation from and proving model properties of SL/SF models. This tool can also show un-reachability of certain model elements.

AutoMOTGen [5] is a non-commercial tool based on model checking. REDIRECT [1] is a test generation tool which uses a combination of (a) random testing, (b) DART (Directed Automated Random Testing) [6] and (c) Hybrid concolic testing [7].

III. SmartTestGen – Test generation techniques

In the following, we will discuss the individual test case generation techniques, which we have integrated within the `STGen`.

A. Model Checking based test generation

Models restricted to a subset of SL/SF are translated into SAL [8]. Keeping a coverage criterion over the SL/SF model in mind, the SAL model is instrumented with *trap variables* such that, the reachability of a trap variable implies reachability of a model element; reachable traces then become the test cases [5]. This technique can also prove un-reachability of certain model elements.

B. Random Testing

Given the input types and their ranges, random test sequences are produced and simulated on the model to check if they cover model elements. The size of each sequence and the number of sequences clearly affect the model coverage.

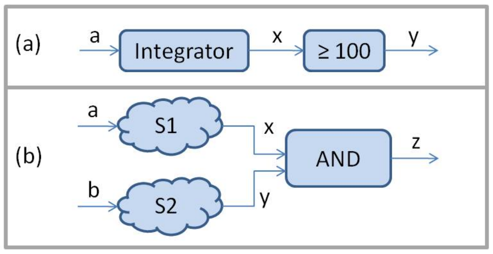

Fig. 1. Example models to illustrate local constraint solving and heuristics.

C. Local Constraint Solving

In order to cover a target – a decision or branching point – in a model, we can take a backward slice of the variables used at the target to determine the relevant internal and external input variables and to obtain a constraint which can be solved to find a test sequence. Constraint solving can also be mixed with random testing.

Figure 1(a) shows a small Simulink model with a discrete integrator. Let x_i denotes the value of variable x at time point i with $i \geq 1$. Assume the initial value of the integrator, i.e., x_0 be $zero$, and inputs are from the range 0..2. Then we have the following equations: for $i \geq 1, x_i = x_{i-1} + a_i$; $y_i = true$ when $x_i \geq 100$, otherwise it is false. Assume, we simulate the model with the random input sequence $[a_1, \ldots, a_{100}]$, and, as a result, we have: $x_{100} = 90$ and $y_{100} = false$. let us apply the symbolic input sequence $[a'_1, \ldots a'_{10}]$ immediately after the earlier random sequence. Then we have the constraint:

$$x'_1 = x_{100} + a'_1 \wedge\ x'_2 = x'_1 + a'_2 \wedge\ \ldots$$
$$x'_{10} = x'_9 + a'_{10} \wedge\ (x'_1 \geq 100 \vee \ldots \vee x'_{10} \geq 100)$$

Let the above constraint be satisfiable, and we get the values for $[a'_1, \ldots a'_{10}]$. Suppose we replace symbolic variables by their concrete values, the new test sequence $[a_1, \ldots, a_{100}]\#[a'_1, \ldots a'_{10}]$ – $\#$ being the concatenation operator – naturally makes the output y true. Here, we have used a combination of random and constraint solving to cover the target. Since we use constraint solving not from the initial state but with respect to some point of an existing trace, we call it *local constraint solving*. Random testing and constraint solving can also be interleaved as in `hybrid concolic testing` [7] to cover deep targets within SL/SF models.

D. Heuristics based Guided Coverage

Figure 1(b) represents a Simulink model, in which S_1 and S_2 are Simulink subsystems. Assume a random input sequence – or such a sequence obtained by constraint solving – $[(a_1, b_1), \ldots (a_{10}, b_{10})]$ makes $x = true$. Similarly, assume there exists a test sequence $[(\overline{a_1}, \overline{b_1}), \ldots (\overline{a_{10}}, \overline{b_{10}})]$ which made $y = true$. Observe that the input sets which compute x and y respectively are disjoint. With this knowledge, we apply our heuristics and derive the input sequence $[(a_1, \overline{b_1}), \ldots (a_{10}, \overline{b_{10}})]$ which makes the output of the AND block true.

Our heuristics library is equipped with a set of heuristics which analyses the patterns of the targets and invoke appropriate heuristics to find test sequences to cover the respective targets. Further details on heuristics can be found in [1].

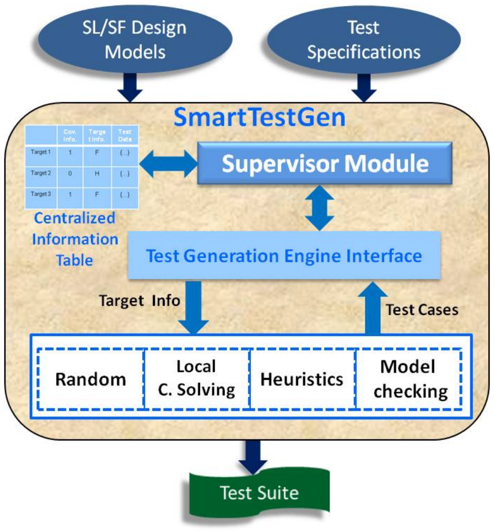

Fig. 2. SmartTestGen Tool Architecture

TABLE I
CENTRALIZED INFORMATION TABLE

Block Handle.	D Cov.	C Cov.	MC/DC Cov.	Block Path	Block Category	Test Vector
BH_1	0/0	2/4	0/2	Path	H	{{..},..}
...	...	...	...	...	...	...
BH_n	...	...	...	...	...	...

IV. SMARTTESTGEN IMPLEMENTATION

Figure 2 outlines the architecture of `STGen` which takes a SL/SF model and a test specification as inputs and produces a test suite as the output. We use the four test generation engines discussed in the previous section. `STGen` has three major components a) Centralized Information table b) a set of test generation engines, and (c) the Supervisor module. The Supervisor module invokes the appropriate test case generation engines, receives the test cases obtained by various engines, and updates the Centralized information table.

A. Centralized Information Table

The Centralized information table (CI_Tab) is designed to contain all the data required to invoke the test generation engines for effective test generation. This table is maintained by the Supervisor module. Table I outlines its structure.

The first column of the table is populated with the block handles – unique identification number – for all the applicable blocks in the model; essentially these blocks contain candidates for coverages. The following three columns respectively contain the decision (D), condition (C) and MC/DC coverage metrics. Each block in the table may contain many number of coverage points, referred to as targets. For example, *Logic* and *Relational* blocks are measured only for decision coverage and have two targets each. Column 5 contains the path of the particular block; a path is a string which identifies the position of a particular block in the hierarchy. A path signifies the

```
Inputs: MD // Given SL/SF model
        Tspec // Test specification given by user
Outputs: TS // Test suite
        UnTar // Unreachable targets
        CI_Tab ← initializeCovTable(MD,Tspec);
        MD1 ← instrumentModel(MD);
        TS1 ← useRandomTestingEngine(MD1);
        CI_Tab ← updateCovTable(MD1,CI_Tab, TS1);
        CI_Tab ← classifyBlocks(MD1,CI_Tab, user_spec);
        Let $TE$ be set of test gen. engines excluding random;
        while $TE \neq \emptyset$ {
                $E$ ← selectEngine(MD1,CI_Tab,TE);
                TS2 ← useEngine(MD1,E, TS1, CI_Tab);
                CI_Tab ← updateCoveTable(MD1,CI_Tab, TS2);
                CI_Tab ← reClassifyBlocks(MD1,CI_Tab);
                TS1 ← updateTestSuite(TS1, TS2);
                $TE \leftarrow TE - \{E\}$; } // end while
        unTar ← MCforUnreachability(MD, CI_Tab);
```

Fig. 3. Functionality of the Supervisor Module

nesting level of a block which is also used to identify which technique would be used to cover this block. Column 7 lists the set of test cases that covers the associated block.

Given a set of SL/SF blocks and their context in the model, a classification algorithm statically estimates which test generation engine(s) is (are) likely to cover which targets; this is determined by using a rule set obtained from empirical knowledge. The user has also the option of overriding this rule set. Column 6 of the table stores this information.

B. Test generation engines

In STGen, we have used three test generation engines which essentially implement the four techniques discussed in the previous section. A random test generation engine generates random input sequences based on the input types and their ranges. A SAL based model checking engine is used to cover given targets by using model checking. A constraint solving and heuristics engine extends a set of given test cases by using local constraint solving and heuristics to cover a set of given targets.

C. Supervisor Module

This module controls the test generation process by invoking the different test generation engines. Figure 3 outlines the main activity of the Supervisor Module.

Routine initializeCovTable() takes the given SL/SF model, finds the targets in the model as per the test specification, and initializes the entries in CI_Tab. Next, routine instrumentModel() instruments the model; instrumentation is required to capture all information associated with a simulation run. Depending on the external inputs and their ranges, next a set of random input sequences are obtained (Test suite TS1). The instrumented model is then simulated with these test cases which results in some coverage. Routine updateCovTable() updates the CI_Tab accordingly. Routine classifyBlocks() fills up the Column 6 of the CI_tab as discussed earlier in this Section.

Depending on the nature of the uncovered targets in CI_Tab, routine selctEngine() selects the appropriate test generation engine from the set of the unused engines. The information in Column 6 of the CI_Tab is used for this selection in the sense that the engine which could cover maximum targets is selected. The selected engine is then used to cover targets. The current test generation engine produces some new test cases (TS2); the coverage table and the current test suite are updated accordingly. Since some of the block targets are covered, Column 6 of the CI_Tab is updated to reflect this fact (routine reClassifyBlocks()). Then the next appropriate test generation engine is selected, and this step is repeated till all engines are used. Finally the model checking engine is invoked again to find if the uncovered targets can be proven to be unreachable. At the end of the procedure, we have the test cases for all the covered targets and the set of unreachable targets.

Currently, we use three test generation engines: (a) the random testing engine, (b) the constraint solving and heuristics based engine, and (c) the model checking engine. Constraint solving and Heuristics techniques are integrated because the former is required to be invoked from within the latter to carry out heuristic steps. At present these engines are invoked in a static order – (i) random, (ii) heuristics based engine and (iii) model checking.

V. Experimental Results

STGen has been implemented using the Matlab scripting language *m-script*. Twenty SL/SF design models from various domains of automotive engineering such as Active safety (AS), Performance traction control (PTC), Powertrain (PT), Heating ventilation and cooling (HVAC) and Electronic stability control (ESC) were used to evaluate the performance of *STGen*. The model sizes vary from 37 blocks to 901 SL/SF blocks. These models contain Stateflow blocks, multi-dimensional inputs, legacy code, non-linear blocks like multiplication and division, dynamic lookup tables and hierarchical triggering of blocks. The most widely used structural coverage criteria like *Condition*, *Decision* and *MC/DC* were given as the test specification for this experimentation. We have used Simulink Verification & Validation (V & V) tool box [9] as our common measuring platform. All the experiments are carried out on a machine with Intel Xeon 3 GHz and 3.5 GB RAM running Windows XP professional. The tool versions used are: BTC Embedded Tester 2.7, Reactis 2009.2 and Matlab R2010a.

We have compared the results of STGen with those of Reactis and Embedded Tester (ET). The comparison results are shown as graphs in Figure 4. The graphs (a), (b) and (c) respectively show the decision, condition and MC/DC coverages of each model. For each model, the first bar shows the coverage of STGen, the second bar shows the cumulative coverage of the random testing and the constraint solving and heuristics engines. Let us refer to the combination of both the above engines as $EngA$. The third bar shows the

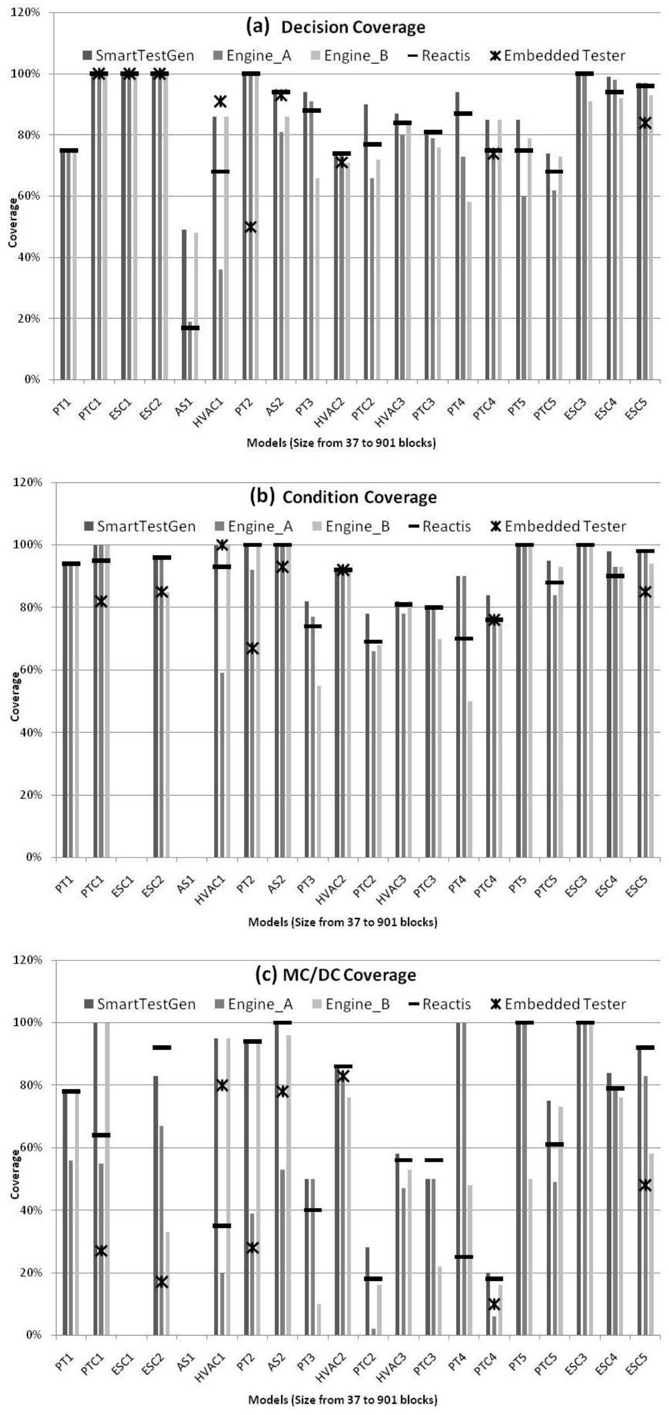

Fig. 4. Comparison of `STGen` results with those of Reactis and ET for (a) Decision coverage, (b) Condition coverage and (c) MC/DC Coverage

coverage obtained by the `model checking` engine, referred to as $EngB$. The Reactis coverage has been shown by a horizontal line, and the the ET coverage by a star symbol. Naturally $Cov(STGen) = Cov(EngA) \cup Cov(EngB)$.

Upon analysis of the results for the three coverage metrics, in 43% of the cases, *EngA* and *EngB* complement each other in the sense that $|cov(STGen)| > |cov(EngA)|$ and $|cov(STGen)| > |cov(EngB)|$, where $||$ denotes the set cardinality operator. For 21% of the cases $cov(EngA) = cov(EngB) = cov(STGen)$, and for the remaining cases, the coverage by one engine subsumes the other.

As regards to the decision coverage, for 11 models *STGen* achieves higher coverage than *Reactis*, while in the remaining cases the coverages obtained are equal. Almost similar phenomena is observed in case of condition coverage. As far as MC/DC is concerned, `STGen` is superior to Reactis in 10 cases, coverages are equal in 6 cases, and in 2 cases Reactis is marginally superior.

Only for a single model, ET outperformed `STGen` for decision coverage. In all other cases, `STGen` results were either superior or equal to ET results.

We have verified that the decision coverage of three models *PT1*, *HVAC2* and *PTC3* shown in Figure 4 are already saturated. Using the model checking technique of *STGen*, we have derived that the remaining targets are un-reachable. This advantage is also observed in case of models *PT1*, *HVAC2* and *PTC3* for condition coverage, and in case of models *PT1* and *HVAC2* for MC/DC coverage.

In our experimentation we observed that *STGen* consumes more time than both *Reactis* and *ET*. In particular, for twelve models *STGen* consumes relatively more time, however, higher coverage was achieved for eight of those models.

VI. Conclusion

The main observations of our experiments are: (a) coverage results of individual test generation techniques, in a broader sense, complement each other and provides better results if regulated effectively, (b) un-reachability results enhances the quality of test adequacy, and (c) when models are safety- or business-critical, we believe, additional coverage at the expense of time is worthwhile.

References

[1] Satpathy M, Yeolekar A, Ramesh S. 2008. Randomized Directed Testing (REDIRECT) for Simulink/Stateflow Models, ACM/IEEE International Conference on Embedded Software (EMSOFT'08), Atlanta.
[2] Reactis. Available at: http://www.reactive-systems.com.
[3] Embedded Tester. BTC Embedded Systems AG. Available at: http://www.btc-es.de/.
[4] The Mathworks, Simulink Design Verifier 1: User's Guide, 2007-08.
[5] Gadkari A, Yeolekar A, Suresh J, Ramesh S, Mohalik S, Shashidhar KC. AutoMOTGen: Automatic Model Oriented Test Generator for Embedded Control Systems. Proceedings of the CAV08, 2008; 204-208.
[6] Godefroid P, Klarlund N, Sen K. 2005.DART: Directed Automated Random Testing, In *Proc. of the PLDI'05*, Chicago, pp. 213-223.
[7] Majumdar, R and Sen, K. Hybrid Concolic Testing, In *Proceedings of the 29th ICSE*, Washington, DC, USA, pp. 416-426.
[8] SRI International. SAL home page http://sal.csl.sri.com
[9] The Mathworks, Verification and Validation Tool Box. Available at: http://www.mathworks.com/verification-validation/.